DFAs in OCAML

We describe here our implementation of DFAs in OCAML. The files that implement this are in the ocaml folder, namely dfa.ml and dfa.mli.

Let's start by the interface:

```
exception InvalidDFA

module type DFA =
  sig
    module A : Alphabet.A
    type state = int
    type elem
    type str
    (* The dfa type *)
    type t

    val make : int -> (state -> elem -> state) -> state list -> t

    val delta : t -> state -> elem -> state
    val deltaStar : t -> state -> str -> state
    val accept : t -> str -> bool

    (* Returns the accepted strings of at most given length *)
    val acceptedStrings : t -> int -> str list

    val union : t -> t -> t
    val intersect : t -> t -> t
  end

module Make(A : Alphabet.A) : DFA with type elem = A.elem
                                  and type str = A.t
```

The module type DFA describes a DFA on a prescribed alphabet A. It introduces a number of types: one for states, represented simply as integers, one for elements of the alphabet, one for strings from the alphabet, and finally a type t to represent a dfa.

The first key method is make, which creates a new dfa. It takes 3 inputs: First the number of states, then a transition function that given a "state" and an element returns a new state, and finally a list of "final states". It returns a dfa (doing some validation first). By convention, the state corresponding to the number 0 is automatically treated as the start state, so no need to specify it.

Following are methods allowing us to trace the accepting of strings: delta carries out one step of the transition function, deltaStar carries out a whole sequence of steps, and accept determines whether the string is accepted by the dfa.

Lastly, acceptedStrings returns all strings of length up to a given integer that are accepted by the dfa.

Finally, two methods implement the construction of the union and intersection of dfas, that we will be discussing in class.

The implementation of dfa.ml is for the most part straightforward. We represent dfas as a *record type*, which we haven't talked about before but should be straightforward:

```
type t = {
    nstates : state;
    delta : state -> elem -> state;
    final : state list;
}
```

The function make essentially just wraps its 3 arguments into an object of type t, and validates it first before returning it (to make sure that the transition function does not take you out of the valid state range, for example, and that the valid states are actually valid states).

The delta function literally just returns the value stored in delta. It is worth noting this expression:

```
let delta { nstates; delta; final } = delta
```

The part { nstates; delta; final } is basically a pattern matching a record. It would normally be written as: { nstates = nstates; delta = delta; final = final } where we use the field names also as variable names. The above is a shorthand for that.

Next up is deltaStar, which is supposed to follow the transition function through a list of inputs. A simple List.fold_left does this nicely.

Then here is accept:

```
let accept ({ nstates; delta; final } as dfa) es =
    List.mem (deltaStar dfa 0 es) final
```

Note the expression ({ nstates; delta; final } as dfa). This says that the first argument should match a record, and bind the 3 arguments to the variables nstates, delta and final, but that the whole argument should also be bound to the variable dfa. All the function does then is use deltaStar to follow the string's steps, starting from the start state 0, and check whether the resulting state is one of the final states.

Lastly, acceptedStrings generates all lists up to a given length, using A.allStringsLeq, then uses List.filter to only keep those that pass the dfa's accept.